Music Player Daemon

Release 0.22.11

Max Kellermann

Aug 24, 2021

CONTENTS:

CHAPTER

ONE

USER'S MANUAL

1.1 Introduction

Music Player Daemon (**MPD**) is a flexible, powerful, server-side application for playing music. Through plugins and libraries it can play a variety of sound files while being controlled by its network protocol.

This document is work in progress. Most of it may be incomplete yet. Please help!

1.2 Installation

1.2.1 Installing on Debian/Ubuntu

Install the package `mpd` via **apt**:

```
apt install mpd
```

When installed this way, **MPD** by default looks for music in `/var/lib/mpd/music/`; this may not be correct. Look at your `/etc/mpd.conf` file...

Note: Debian and Ubuntu are infamous for shipping heavily outdated software. The **MPD** version in their respective stable releases are usually too old to be supported by this project. Ironically, the **MPD** version in Debian "*unstable*" is more stable than the version in Debian "*stable*".

1.2.2 Installing on Android

An experimental Android build is available on Google Play. After installing and launching it, **MPD** will scan the music in your Music directory and you can control it as usual with a **MPD** client.

If you need to tweak the configuration, you can create a file called `mpd.conf` on the data partition (the directory which is returned by Android's *getExternalStorageDirectory()* API function).

ALSA is not available on Android; only the *OpenSL ES* output plugin can be used for local playback.

1.2.3 Compiling from source

Download the source tarball and unpack it (or clone the git repository):

```
tar xf mpd-version.tar.xz
cd mpd-version
```

In any case, you need:

- a C++17 compiler (e.g. GCC 8 or clang 7)
- Meson 0.49.0 and Ninja
- Boost 1.58
- pkg-config

Each plugin usually needs a codec library, which you also need to install. Check the *Plugin reference* for details about required libraries

For example, the following installs a fairly complete list of build dependencies on Debian Buster:

```
apt install meson g++ \
  libpcre3-dev \
  libmad0-dev libmpg123-dev libid3tag0-dev \
  libflac-dev libvorbis-dev libopus-dev libogg-dev \
  libadplug-dev libaudiofile-dev libsndfile1-dev libfaad-dev \
  libfluidsynth-dev libgme-dev libmikmod-dev libmodplug-dev \
  libmpcdec-dev libwavpack-dev libwildmidi-dev \
  libsidplay2-dev libsidutils-dev libresid-builder-dev \
  libavcodec-dev libavformat-dev \
  libmp3lame-dev libtwolame-dev libshine-dev \
  libsamplerate0-dev libsoxr-dev \
  libbz2-dev libcdio-paranoia-dev libiso9660-dev libmms-dev \
  libzzip-dev \
  libcurl4-gnutls-dev libyajl-dev libexpat-dev \
  libasound2-dev libao-dev libjack-jackd2-dev libopenal-dev \
  libpulse-dev libshout3-dev \
  libsndio-dev \
  libmpdclient-dev \
  libnfs-dev \
  libupnp-dev \
  libavahi-client-dev \
  libsqlite3-dev \
  libsystemd-dev \
  libgtest-dev \
  libboost-dev \
  libicu-dev \
  libchromaprint-dev \
  libgcrypt20-dev
```

Now configure the source tree:

```
meson . output/release --buildtype=debugoptimized -Db_ndebug=true
```

The following command shows a list of compile-time options:

```
meson configure output/release
```

When everything is ready and configured, compile:

```
ninja -C output/release
```

And install:

```
ninja -C output/release install
```

1.2.4 Compiling for Windows

Even though it does not "feel" like a Windows application, **MPD** works well under Windows. Its build process follows the "Linux style" and may seem awkward for Windows people (who are not used to compiling their software, anyway).

Basically, there are two ways to compile **MPD** for Windows:

- Build as described above: with **meson** and **ninja**. To cross-compile from Linux, you need a Meson cross file.

  The remaining difficulty is installing all the external libraries. And **MPD** usually needs many, making this method cumbersome for the casual user.

- Build on Linux for Windows using **MPD**'s library build script.

This section is about the latter.

You need:

- mingw-w64
- Meson 0.49.0 and Ninja
- cmake
- pkg-config
- quilt

Just like with the native build, unpack the **MPD** source tarball and change into the directory. Then, instead of **meson**, type:

```
mkdir -p output/win64
cd output/win64
../../win32/build.py --64
```

This downloads various library sources, and then configures and builds **MPD** (for x64; to build a 32 bit binary, pass `--32`). The resulting EXE files is linked statically, i.e. it contains all the libraries already and you do not need carry DLLs around. It is large, but easy to use. If you wish to have a small mpd.exe with DLLs, you need to compile manually, without the `build.py` script.

1.2.5 Compiling for Android

MPD can be compiled as an Android app. It can be installed easily with Google Play, but if you want to build it from source, follow this section.

You need:

- Android SDK
- Android NDK r23
- Meson 0.49.0 and Ninja
- cmake
- pkg-config
- quilt

Just like with the native build, unpack the **MPD** source tarball and change into the directory. Then, instead of **meson**, type:

```
mkdir -p output/android
cd output/android
../../android/build.py SDK_PATH NDK_PATH ABI
meson configure -Dandroid_debug_keystore=$HOME/.android/debug.keystore
ninja android/apk/mpd-debug.apk
```

`SDK_PATH` is the absolute path where you installed the Android SDK; `NDK_PATH` is the Android NDK installation path; ABI is the Android ABI to be built, e.g. "`arm64-v8a`".

This downloads various library sources, and then configures and builds **MPD**.

1.3 Configuration

1.3.1 The Configuration File

MPD reads its configuration from a text file. Usually, that is `/etc/mpd.conf`, unless a different path is specified on the command line. If you run **MPD** as a user daemon (and not as a system daemon), the configuration is read from `$XDG_CONFIG_HOME/mpd/mpd.conf` (usually `~/.config/mpd/mpd.conf`). On Android, `mpd.conf` will be loaded from the top-level directory of the data partition.

Each line in the configuration file contains a setting name and its value, e.g.:

```
connection_timeout "5"
```

For settings which specify a filesystem path, the tilde is expanded:

```
music_directory "~/Music"
```

Some of the settings are grouped in blocks with curly braces, e.g. per-plugin settings:

```
audio_output {
    type "alsa"
    name "My ALSA output"
    device "iec958:CARD=Intel,DEV=0"
    mixer_control "PCM"
}
```

The `include` directive can be used to include settings from another file; the given file name is relative to the current file:

```
include "other.conf"
```

You can use `include_optional` instead if you want the included file to be optional; the directive will be ignored if the file does not exist:

```
include_optional "may_not_exist.conf"
```

1.3.2 Configuring the music directory

When you play local files, you should organize them within a directory called the "music directory". This is configured in **MPD** with the music_directory setting.

By default, **MPD** follows symbolic links in the music directory. This behavior can be switched off: `follow_outside_symlinks` controls whether **MPD** follows links pointing to files outside of the music directory, and `follow_inside_symlinks` lets you disable symlinks to files inside the music directory.

Instead of using local files, you can use storage plugins to access files on a remote file server. For example, to use music from the SMB/CIFS server "`myfileserver`" on the share called "Music", configure the music directory "`smb://myfileserver/Music`". For a recipe, read the Satellite **MPD** section *Satellite setup*.

You can also use multiple storage plugins to assemble a virtual music directory consisting of multiple storages.

1.3.3 Configuring database plugins

If a music directory is configured, one database plugin is used. To configure this plugin, add a `database` block to `mpd.conf`:

```
database {
    plugin "simple"
    path "/var/lib/mpd/db"
}
```

More information can be found in the *Database plugins* reference.

1.3.4 Configuring neighbor plugins

All neighbor plugins are disabled by default to avoid unwanted overhead. To enable (and configure) a plugin, add a `neighbor` block to `mpd.conf`:

```
neighbors {
    plugin "smbclient"
}
```

More information can be found in the *Neighbor plugins* reference.

1.3.5 Configuring input plugins

To configure an input plugin, add an `input` block to `mpd.conf`:

```
input {
    plugin "curl"
    proxy "proxy.local"
}
```

The following table lists the input options valid for all plugins:

Name	Description
plugin	The name of the plugin
enabled yes|no	Allows you to disable a input plugin without recompiling. By default, all plugins are enabled.

More information can be found in the *Input plugins* reference.

Configuring the Input Cache

The input cache prefetches queued song files before they are going to be played. This has several advantages:

- risk of buffer underruns during playback is reduced because this decouples playback from disk (or network) I/O
- bulk transfers may be faster and more energy efficient than loading small chunks on-the-fly
- by prefetching several songs at a time, the hard disk can spin down for longer periods of time

This comes at a cost:

- memory usage
- bulk transfers may reduce the performance of other applications which also want to access the disk (if the kernel's I/O scheduler isn't doing its job properly)

To enable the input cache, add an `input_cache` block to the configuration file:

```
input_cache {
    size "1 GB"
}
```

This allocates a cache of 1 GB. If the cache grows larger than that, older files will be evicted.

You can flush the cache at any time by sending `SIGHUP` to the **MPD** process, see *Signals*.

1.3.6 Configuring decoder plugins

Most decoder plugins do not need any special configuration. To configure a decoder, add a `decoder` block to `mpd.conf`:

```
decoder {
    plugin "wildmidi"
    config_file "/etc/timidity/timidity.cfg"
}
```

The following table lists the decoder options valid for all plugins:

Name	Description
plugin	The name of the plugin
enabled yes|no	Allows you to disable a decoder plugin without recompiling. By default, all plugins are enabled.

More information can be found in the *Decoder plugins* reference.

1.3.7 Configuring encoder plugins

Encoders are used by some of the output plugins (such as shout). The encoder settings are included in the `audio_output` section, see *Configuring audio outputs*.

More information can be found in the *Encoder plugins* reference.

1.3.8 Configuring audio outputs

Audio outputs are devices which actually play the audio chunks produced by **MPD**. You can configure any number of audio output devices, but there must be at least one. If none is configured, **MPD** attempts to auto-detect. Usually, this works quite well with ALSA, OSS and on Mac OS X.

To configure an audio output manually, add one or more `audio_output` blocks to `mpd.conf`:

```
audio_output {
    type "alsa"
    name "my ALSA device"
    device "hw:0"
}
```

The following table lists the audio_output options valid for all plugins:

Name	Description
type	The name of the plugin
name	The name of the audio output. It is visible to the client. Some plugins also use it internally, e.g. as a name registered in the PULSE server.
format samplerate:bits:channels	Always open the audio output with the specified audio format, regardless of the format of the input file. This is optional for most plugins. See *Global Audio Format* for a detailed description of the value.
enabled yes|no	Specifies whether this audio output is enabled when **MPD** is started. By default, all audio outputs are enabled. This is just the default setting when there is no state file; with a state file, the previous state is restored.
tags yes|no	If set to no, then **MPD** will not send tags to this output. This is only useful for output plugins that can receive tags, for example the httpd output plugin.
always_on yes|no	If set to yes, then **MPD** attempts to keep this audio output always open. This may be useful for streaming servers, when you don't want to disconnect all listeners even when playback is accidentally stopped.
mixer_type hardware|software|null|none	Specifies which mixer should be used for this audio output: the hardware mixer (available for ALSA *alsa*, OSS *oss* and PulseAudio *pulse*), the software mixer, the "`null`" mixer (allows setting the volume, but with no effect; this can be used as a trick to implement an external mixer, see *External Mixer*) or no mixer (`none`). By default, the hardware mixer is used for devices which support it, and none for the others.
filters "name,..."	The specified configured filters are instantiated in the given order. Each filter name refers to a `filter` block, see *Configuring filters*.

More information can be found in the *Output plugins* reference.

1.3.9 Configuring filters

Filters are plugins which modify an audio stream.

To configure a filter, add a `filter` block to `mpd.conf`:

```
filter {
    plugin "volume"
    name "software volume"
}
```

Configured filters may then be added to the `filters` setting of an `audio_output` section, see *Configuring audio outputs*.

The following table lists the filter options valid for all plugins:

Name	Description
plugin	The name of the plugin
name	The name of the filter

More information can be found in the *Filter plugins* reference.

1.3.10 Configuring playlist plugins

Playlist plugins are used to load remote playlists (protocol commands load, listplaylist and listplaylistinfo). This is not related to **MPD**'s *playlist directory*.

To configure a playlist plugin, add a `playlist_plugin` block to `mpd.conf`:

```
playlist_plugin {
    name "m3u"
    enabled "true"
}
```

The following table lists the playlist_plugin options valid for all plugins:

Name	Description
plugin	The name of the plugin
enabled yes|no	Allows you to disable a playlist plugin without recompiling. By default, all plugins are enabled.
as_directory yes|no	With this option, a playlist file of this type is parsed during database update and converted to a virtual directory, allowing MPD clients to access individual entries. By default, this is only enabled for the *cue plugin*.

More information can be found in the *Playlist plugins* reference.

1.3.11 Audio Format Settings

Global Audio Format

The setting `audio_output_format` forces **MPD** to use one audio format for all outputs. Doing that is usually not a good idea.

The value is specified as `samplerate:bits:channels`.

Any of the three attributes may be an asterisk to specify that this attribute should not be enforced, example: `48000:16:*`. `*:*:*` is equal to not having a format specification.

The following values are valid for bits: `8` (signed 8 bit integer samples), `16`, `24` (signed 24 bit integer samples padded to 32 bit), `32` (signed 32 bit integer samples), `f` (32 bit floating point, -1.0 to 1.0), `dsd` means DSD (Direct Stream Digital). For DSD, there are special cases such as `dsd64`, which allows you to omit the sample rate (e.g. `dsd512:2` for stereo DSD512, i.e. 22.5792 MHz).

The sample rate is special for DSD: **MPD** counts the number of bytes, not bits. Thus, a DSD "bit" rate of 22.5792 MHz (DSD512) is 2822400 from **MPD**'s point of view (44100*512/8).

Resampler

Sometimes, music needs to be resampled before it can be played; for example, CDs use a sample rate of 44,100 Hz while many cheap audio chips can only handle 48,000 Hz. Resampling reduces the quality and consumes a lot of CPU. There are different options, some of them optimized for high quality and others for low CPU usage, but you can't have both at the same time. Often, the resampler is the component that is responsible for most of **MPD**'s CPU usage. Since **MPD** comes with high quality defaults, it may appear that **MPD** consumes more CPU than other software.

Check the *Resampler plugins* reference for a list of resamplers and how to configure them.

1.3.12 Client Connections

Listeners

The setting `bind_to_address` specifies which addresses **MPD** listens on for connections from clients. It can be used multiple times to bind to more than one address. Example:

```
bind_to_address "192.168.1.42"
bind_to_address "127.0.0.1"
```

The default is "any", which binds to all available addresses. Additionally, MPD binds to `$XDG_RUNTIME_DIR/mpd/socket` (if it was launched as a per-user daemon and no `bind_to_address` setting exists).

You can set a port that is different from the global port setting, e.g. "localhost:6602". IPv6 addresses must be enclosed in square brackets if you want to configure a port:

```
bind_to_address "[::1]:6602"
```

To bind to a local socket (UNIX domain socket), specify an absolute path or a path starting with a tilde (~). Some clients default to connecting to `/var/run/mpd/socket` so this may be a good choice:

```
bind_to_address "/var/run/mpd/socket"
```

On Linux, local sockets can be bound to a name without a socket inode on the filesystem; MPD implements this by prepending `@` to the address:

```
bind_to_address "@mpd"
```

If no port is specified, the default port is 6600. This default can be changed with the port setting:

```
port "6601"
```

These settings will be ignored if *systemd socket activation* is used.

Permissions and Passwords

By default, all clients are unauthenticated and have a full set of permissions. This can be restricted with the settings `default_permissions` and `password`.

`default_permissions` controls the permissions of a new client. Its value is a comma-separated list of permissions:

Name	Description
read	Allows reading of the database, displaying the current playlist, and current status of **MPD**.
add	Allows adding songs and loading playlists.
control	Allows all other player and playlist manipulations.
admin	Allows manipulating outputs, stickers and partitions, mounting/unmounting storage and shutting down **MPD**.

`local_permissions` may be used to assign other permissions to clients connecting on a local socket.

`password` allows the client to send a password to gain other permissions. This option may be specified multiple times with different passwords.

Note that the `password` option is not secure: passwords are sent in clear-text over the connection, and the client cannot verify the server's identity.

Example:

```
default_permissions "read"
password "the_password@read,add,control"
password "the_admin_password@read,add,control,admin"
```

1.3.13 Other Settings

Setting	Description
metadata_to_use TAG1,TAG2,...	Use only the specified tags, and ignore the others. This setting can reduce the database size and **MPD**'s memory usage by omitting unused tags. By default, all tags but comment are enabled. The special value "none" disables all tags. If the setting starts with + or -, then the following tags will be added or remoted to/from the current set of tags. This example just enables the "comment" tag without disabling all the other supported tags metadata_to_use "+comment" Section *Tags* contains a list of supported tags.

The State File

The state file is a file where **MPD** saves and restores its state (play queue, playback position etc.) to keep it persistent across restarts and reboots. It is an optional setting.

MPD will attempt to load the state file during startup, and will save it when shutting down the daemon. Additionally, the state file is refreshed every two minutes (after each state change).

Setting	Description
state_file PATH	Specify the state file location. The parent directory must be writable by the **MPD** user (+wx).
state_file_interval SECONDS	Auto-save the state file this number of seconds after each state change. Defaults to 120 (2 minutes).
restore_paused yes|no	If set to `yes`, then **MPD** is put into pause mode instead of starting playback after startup. Default is `no`.

The Sticker Database

"Stickers" are pieces of information attached to songs. Some clients use them to store ratings and other volatile data. This feature requires **SQLite**, compile-time configure option `-Dsqlite=...`.

Setting	Description
sticker_file PATH	The location of the sticker database.

Resource Limitations

These settings are various limitations to prevent **MPD** from using too many resources (denial of service).

Setting	Description
connection_timeout SECONDS	If a client does not send any new data in this time period, the connection is closed. Clients waiting in "idle" mode are excluded from this. Default is 60.
max_connections NUMBER	This specifies the maximum number of clients that can be connected to **MPD** at the same time. Default is 100.
max_playlist_length NUMBER	The maximum number of songs that can be in the playlist. Default is 16384.
max_command_list_size KBYTES	The maximum size a command list. Default is 2048 (2 MiB).
max_output_buffer_size KBYTES	The maximum size of the output buffer to a client (maximum response size). Default is 8192 (8 MiB).

Buffer Settings

Do not change these unless you know what you are doing.

Setting	Description
audio_buffer_size SIZE	Adjust the size of the internal audio buffer. Default is `4 MB` (4 MiB).

Zeroconf

If Zeroconf support (Avahi or Apple's Bonjour) was enabled at compile time with `-Dzeroconf=...`, **MPD** can announce its presence on the network. The following settings control this feature:

Setting	Description
zeroconf_enabled yes|no	Enables or disables this feature. Default is yes.
zeroconf_name NAME	The service name to publish via Zeroconf. The default is "Music Player @ %h". %h will be replaced with the hostname of the machine running **MPD**.

1.4 Advanced configuration

1.4.1 Satellite setup

MPD runs well on weak machines such as the Raspberry Pi. However, such hardware tends to not have storage big enough to hold a music collection. Mounting music from a file server can be very slow, especially when updating the database.

One approach for optimization is running **MPD** on the file server, which not only exports raw files, but also provides access to a readily scanned database. Example configuration:

```
music_directory "nfs://fileserver.local/srv/mp3"
#music_directory "smb://fileserver.local/mp3"

database {
    plugin "proxy"
    host "fileserver.local"
}
```

The `music_directory` setting tells **MPD** to read files from the given NFS server. It does this by connecting to the server from userspace. This does not actually mount the file server into the kernel's virtual file system, and thus requires no kernel cooperation and no special privileges. It does not even require a kernel with NFS support, only the nfs storage plugin (using the libnfs userspace library). The same can be done with SMB/CIFS using the smbclient storage plugin (using libsmbclient).

The database setting tells **MPD** to pass all database queries on to the **MPD** instance running on the file server (using the proxy plugin).

1.4.2 Real-Time Scheduling

On Linux, **MPD** attempts to configure real-time scheduling for some threads that benefit from it.

This is only possible if you allow **MPD** to do it. This privilege is controlled by `RLIMIT_RTPRIO` `RLIMIT_RTTIME`. You can configure this privilege with **ulimit** before launching **MPD**:

```
ulimit -HS -r 40; mpd
```

Or you can use the **prlimit** program from the util-linux package:

```
prlimit --rtprio=40 --rttime=unlimited mpd
```

The systemd service file shipped with **MPD** comes with this setting.

This works only if the Linux kernel was compiled with **CONFIG_RT_GROUP_SCHED** disabled. Use the following command to check this option for your current kernel:

```
zgrep ^CONFIG_RT_GROUP_SCHED /proc/config.gz
```

You can verify whether the real-time scheduler is active with the ps command:

```
# ps H -q `pidof -s mpd` -o 'pid,tid,cls,rtprio,comm'
  PID   TID CLS RTPRIO COMMAND
16257 16257  TS      - mpd
16257 16258  TS      - io
16257 16259  FF     40 rtio
16257 16260  TS      - player
16257 16261  TS      - decoder
16257 16262  FF     40 output:ALSA
16257 16263 IDL      0 update
```

The CLS column shows the CPU scheduler; TS is the normal scheduler; FF and RR are real-time schedulers. In this example, two threads use the real-time scheduler: the output thread and the rtio (real-time I/O) thread; these two are the important ones. The database update thread uses the idle scheduler ("IDL in ps), which only gets CPU when no other process needs it.

Note: There is a rumor that real-time scheduling improves audio quality. That is not true. All it does is reduce the probability of skipping (audio buffer xruns) when the computer is under heavy load.

1.5 Using MPD

1.5.1 Starting and Stopping MPD

The simplest (but not the best) way to start **MPD** is to simply type:

```
mpd
```

This will start **MPD** as a daemon process (which means it detaches from your terminal and continues to run in background). To stop it, send `SIGTERM` to the process; if you have configured a `pid_file`, you can use the `--kill` option:

```
mpd --kill
```

The best way to manage **MPD** processes is to use a service manager such as **systemd**.

systemd

MPD ships with **systemd** service units.

If you have installed **MPD** with your operating system's package manager, these are probably preinstalled, so you can start and stop **MPD** this way (like any other service):

```
systemctl start mpd
systemctl stop mpd
```

systemd socket activation

Using systemd, you can launch **MPD** on demand when the first client attempts to connect.

MPD comes with two systemd unit files: a "service" unit and a "socket" unit. These will be installed to the directory specified with `-Dsystemd_system_unit_dir=...`, e.g. `/lib/systemd/system`.

To enable socket activation, type:

```
systemctl enable mpd.socket
systemctl start mpd.socket
```

In this configuration, **MPD** will ignore the *listener settings* (`bind_to_address` and `port`).

systemd user unit

You can launch **MPD** as a systemd user unit. These will be installed to the directory specified with `-Dsystemd_user_unit_dir=...`, e.g. `/usr/lib/systemd/user` or `$HOME/.local/share/systemd/user`.

Once the user unit is installed, you can start and stop **MPD** like any other service:

```
systemctl --user start mpd
```

To auto-start **MPD** upon login, type:

```
systemctl --user enable mpd
```

1.5.2 Signals

MPD understands the following UNIX signals:

- `SIGTERM`, `SIGINT`: shut down MPD
- `SIGHUP`: reopen log files (send this after log rotation) and flush caches (see *Configuring the Input Cache*)

1.5.3 The client

After you have installed, configured and started **MPD**, you choose a client to control the playback.

The most basic client is **mpc**, which provides a command line interface. It is useful in shell scripts. Many people bind specific **mpc** commands to hotkeys.

The MPD Wiki contains an extensive list of clients to choose from.

1.5.4 The music directory and the database

The "music directory" is where you store your music files. **MPD** stores all relevant meta information about all songs in its "database". Whenever you add, modify or remove songs in the music directory, you have to update the database, for example with mpc:

```
mpc update
```

Depending on the size of your music collection and the speed of the storage, this can take a while.

To exclude a file from the update, create a file called `.mpdignore` in its parent directory. Each line of that file may contain a list of shell wildcards. Matching files in the current directory and all subdirectories are excluded.

Mounting other storages into the music directory

MPD has various storage plugins of which multiple instances can be "mounted" into the music directory. This way, you can use local music, file servers and USB sticks at the same time. Example:

```
mpc mount foo nfs://192.168.1.4/export/mp3
mpc mount usbstick udisks://by-uuid-2F2B-D136
mpc unmount usbstick
```

MPD's neighbor plugins can be helpful with finding mountable storages:

```
mpc listneighbors
```

Mounting is only possible with the simple database plugin and a `cache_directory`, e.g.:

```
database {
  plugin "simple"
  path "~/.mpd/db"
  cache_directory "~/.mpd/cache"
}
```

This requires migrating from the old `db_file` setting to a database section. The cache directory must exist, and **MPD** will put one file per mount there, which will be reused when the same storage is used again later.

1.5.5 Metadata

When scanning or playing a song, **MPD** parses its metadata. See *Tags* for a list of supported tags.

The *metadata_to_use* setting can be used to enable or disable certain tags.

1.5.6 The queue

The queue (sometimes called "current playlist") is a list of songs to be played by **MPD**. To play a song, add it to the queue and start playback. Most clients offer an interface to edit the queue.

1.5.7 Stored Playlists

Stored playlists are some kind of secondary playlists which can be created, saved, edited and deleted by the client. They are addressed by their names. Its contents can be loaded into the queue, to be played back. The `playlist_directory` setting specifies where those playlists are stored.

1.6 Advanced usage

1.6.1 Bit-perfect playback

"Bit-perfect playback" is a phrase used by audiophiles to describe a setup that plays back digital music as-is, without applying any modifications such as resampling, format conversion or software volume. Naturally, this implies a lossless codec.

By default, **MPD** attempts to do bit-perfect playback, unless you tell it not to. Precondition is a sound chip that supports the audio format of your music files. If the audio format is not supported, **MPD** attempts to fall back to the nearest supported audio format, trying to lose as little quality as possible.

To verify if **MPD** converts the audio format, enable verbose logging, and watch for these lines:

```
decoder: audio_format=44100:24:2, seekable=true
output: opened plugin=alsa name="An ALSA output" audio_format=44100:16:2
output: converting from 44100:24:2
```

This example shows that a 24 bit file is being played, but the sound chip cannot play 24 bit. It falls back to 16 bit, discarding 8 bit.

However, this does not yet prove bit-perfect playback; ALSA may be fooling **MPD** that the audio format is supported. To verify the format really being sent to the physical sound chip, try:

```
cat /proc/asound/card*/pcm*p/sub*/hw_params
access: RW_INTERLEAVED
format: S16_LE
subformat: STD
channels: 2
rate: 44100 (44100/1)
period_size: 4096
buffer_size: 16384
```

Obey the "format" row, which indicates that the current playback format is 16 bit (signed 16 bit integer, little endian).

Check list for bit-perfect playback:

- Use the ALSA output plugin.
- Disable sound processing inside ALSA by configuring a "hardware" device (`hw:0,0` or similar).
- Don't use software volume (setting `mixer_type`).
- Don't force **MPD** to use a specific audio format (settings `format`, *audio_output_format*).
- Verify that you are really doing bit-perfect playback using **MPD**'s verbose log and `/proc/asound/card*/pcm*p/sub*/hw_params`. Some DACs can also indicate the audio format.

1.6.2 Direct Stream Digital (DSD)

DSD (Direct Stream Digital) is a digital format that stores audio as a sequence of single-bit values at a very high sampling rate. It is the sample format used on Super Audio CDs.

MPD understands the file formats *DSDIFF* and *DSF*. There are three ways to play back DSD:

- Native DSD playback. Requires ALSA 1.0.27.1 or later, a sound driver/chip that supports DSD and of course a DAC that supports DSD.
- DoP (DSD over PCM) playback. This wraps DSD inside fake 24 bit PCM according to the DoP standard. Requires a DAC that supports DSD. No support from ALSA and the sound chip required (except for bit-perfect 24 bit PCM support).
- Convert DSD to PCM on-the-fly.

Native DSD playback is used automatically if available. DoP is only used if enabled explicitly using the `dop` option, because there is no way for **MPD** to find out whether the DAC supports it. DSD to PCM conversion is the fallback if DSD cannot be used directly.

1.6.3 ICY-MetaData

Some MP3 streams send information about the current song with a protocol named "ICY-MetaData". **MPD** makes its `StreamTitle` value available as `Title` tag.

By default, **MPD** assumes this tag is UTF-8-encoded. To tell **MPD** to assume a different character set, specify it in the `charset` URL fragment parameter, e.g.:

```
mpc add 'http://radio.example.com/stream#charset=cp1251'
```

1.7 Client Hacks

1.7.1 External Mixer

The setting `mixer_type "null"` asks MPD to pretend that there is a mixer, but not actually do something. This allows you to implement a **MPD** client which listens for mixer events, queries the current (fake) volume, and uses it to program an external mixer. For example, your client can forward this setting to your amplifier.

1.8 Troubleshooting

1.8.1 Where to start

Make sure you have the latest **MPD** version (via `mpd --version`, not mpc version). All the time, bugs are found and fixed, and your problem might be a bug that is fixed already. Do not ask for help unless you have the latest **MPD** version. The most common excuse is when your distribution ships an old **MPD** version - in that case, please ask your distribution for help, and not the **MPD** project.

Check the log file. Configure `log_level "verbose"` or pass `--verbose` to mpd.

Sometimes, it is helpful to run **MPD** in a terminal and follow what happens. This is how to do it:

```
mpd --stderr --no-daemon --verbose
```

1.8.2 Support

Getting Help

The **MPD** project runs a forum and an IRC channel (#mpd on Libera.Chat) for requesting help. Visit the MPD help page for details on how to get help.

Common Problems

Startup

Error “could not get realtime scheduling”

See *Real-Time Scheduling*. You can safely ignore this, but you won’t benefit from real-time scheduling. This only makes a difference if your computer runs programs other than MPD.

Error "Failed to initialize io_uring"

Linux specific: the io_uring subsystem could not be initialized. This is not a critical error - MPD will fall back to "classic" blocking disk I/O. You can safely ignore this error, but you won't benefit from io_uring's advantages.

- "Cannot allocate memory" usually means that your memlock limit (`ulimit -l` in bash or `LimitMEMLOCK` in systemd) is too low. 64 MB is a reasonable value for this limit.
- Your Linux kernel might be too old and does not support io_uring.

Error "bind to '0.0.0.0:6600' failed (continuing anyway, because binding to '[::]:6600' succeeded)"

This happens on Linux when `/proc/sys/net/ipv6/bindv6only` is disabled. MPD first binds to IPv6, and this automatically binds to IPv4 as well; after that, MPD binds to IPv4, but that fails. You can safely ignore this, because MPD works on both IPv4 and IPv6.

Database

I can't see my music in the MPD database

- Check your `music_directory` setting.
- Does the MPD user have read permission on all music files, and read+execute permission on all music directories (and all of their parent directories)?
- Did you update the database? (mpc update)
- Did you enable all relevant decoder plugins at compile time? **`mpd --version`** will tell you.

MPD doesn't read ID3 tags!

- You probably compiled **MPD** without libid3tag. **`mpd --version`** will tell you.

Playback

I can't hear music on my client

- That problem usually follows a misunderstanding of the nature of **MPD**. **MPD** is a remote-controlled music player, not a music distribution system. Usually, the speakers are connected to the box where **MPD** runs, and the **MPD** client only sends control commands, but the client does not actually play your music.

  MPD has output plugins which allow hearing music on a remote host (such as httpd), but that is not **MPD**'s primary design goal.

Error "Device or resource busy"

- This ALSA error means that another program uses your sound hardware exclusively. You can stop that program to allow **MPD** to use it.

  Sometimes, this other program is PulseAudio, which can multiplex sound from several applications, to allow them to share your sound chip. In this case, it might be a good idea for **MPD** to use PulseAudio as well, instead of using ALSA directly.

1.8.3 Reporting Bugs

If you believe you found a bug in **MPD**, report it on the bug tracker.

Your bug report should contain:

- the output of **mpd --version**
- your configuration file (`mpd.conf`)
- relevant portions of the log file (`--verbose`)
- be clear about what you expect MPD to do, and what is actually happening

MPD crashes

All **MPD** crashes are bugs which must be fixed by a developer, and you should write a bug report. (Many crash bugs are caused by codec libraries used by **MPD**, and then that library must be fixed; but in any case, the **MPD** bug tracker is a good place to report it first if you don't know.)

A crash bug report needs to contain a "backtrace".

First of all, your **MPD** executable must not be "stripped" (i.e. debug information deleted). The executables shipped with Linux distributions are usually stripped, but some have so-called "debug" packages (package `mpd-dbgsym` or `mpd-dbg` on Debian, `mpd-debug` on other distributions). Make sure this package is installed.

If you built **MPD** from sources, please recompile with Meson option "`--buildtype=debug -Db_ndebug=false`", because this will add more helpful information to the backtrace.

You can extract the backtrace from a core dump, or by running **MPD** in a debugger, e.g.:

```
gdb --args mpd --stderr --no-daemon --verbose
run
```

As soon as you have reproduced the crash, type "**bt**" on the gdb command prompt. Copy the output to your bug report.

CHAPTER

TWO

PLUGIN REFERENCE

2.1 Database plugins

2.1.1 simple

The default plugin. Stores a copy of the database in memory. A file is used for permanent storage.

Setting	Description
path	The path of the database file.
cache_directory	The path of the cache directory for additional storages mounted at runtime. This setting is necessary for the **mount** protocol command.
compress yes|no	Compress the database file using gzip? Enabled by default (if built with zlib).

2.1.2 proxy

Provides access to the database of another **MPD** instance using libmpdclient. This is useful when you mount the music directory via NFS/SMB, and the file server already runs a **MPD** instance. Only the file server needs to update the database.

Setting	Description
host	The host name of the "master" **MPD** instance.
port	The port number of the "master" **MPD** instance.
password	The password used to log in to the "master" **MPD** instance.
keepalive yes|no	Send TCP keepalive packets to the "master" **MPD** instance? This option can help avoid certain firewalls dropping inactive connections, at the expense of a very small amount of additional network traffic. Disabled by default.

2.1.3 upnp

Provides access to UPnP media servers.

2.2 Storage plugins

2.2.1 local

The default plugin which gives **MPD** access to local files. It is used when music_directory refers to a local directory.

2.2.2 curl

A WebDAV client using libcurl. It is used when `music_directory` contains a `http://` or `https://` URI, for example `https://the.server/dav/`.

2.2.3 smbclient

Load music files from a SMB/CIFS server. It is used when `music_directory` contains a `smb://` URI, for example `smb://myfileserver/Music`.

Note that `libsmbclient` has a serious bug which causes MPD to crash, and therefore this plugin is disabled by default and should not be used until the bug is fixed: https://bugzilla.samba.org/show_bug.cgi?id=11413

2.2.4 nfs

Load music files from a NFS server. It is used when `music_directory` contains a `nfs://` URI according to RFC2224, for example `nfs://servername/path`.

See *nfs* for more information.

2.2.5 udisks

Mount file systems (e.g. USB sticks or other removable media) using the udisks2 daemon via D-Bus. To obtain a valid udisks2 URI, consult *the according neighbor plugin*.

It might be necessary to grant **MPD** privileges to control **udisks2** through **policykit**. To do this, create a file called `/usr/share/polkit-1/rules.d/mpd-udisks.rules` with the following text:

```
polkit.addRule(function(action, subject) {
  if ((action.id == "org.freedesktop.udisks2.filesystem-mount" ||
       action.id == "org.freedesktop.udisks2.filesystem-mount-other-seat") &&
      subject.user == "mpd") {
      return polkit.Result.YES;
  }
});
```

If you run MPD as a different user, change `mpd` to the name of your MPD user.

2.3 Neighbor plugins

2.3.1 smbclient

Provides a list of SMB/CIFS servers on the local network.

2.3.2 udisks

Queries the udisks2 daemon via D-Bus and obtains a list of file systems (e.g. USB sticks or other removable media).

2.3.3 upnp

Provides a list of UPnP servers on the local network.

2.4 Input plugins

2.4.1 alsa

Allows **MPD** on Linux to play audio directly from a soundcard using the scheme alsa://. Audio is by default formatted as 48 kHz 16-bit stereo, but this default can be overidden by a config file setting or by the URI. Examples:

```
mpc add alsa:// plays audio from device default
```

```
mpc add alsa://hw:1,0 plays audio from device hw:1,0
```

```
mpc add alsa://hw:1,0?format=44100:16:2 plays audio from device hw:1,0 sampling 16-bit␣
↪stereo at 44.1kHz.
```

Setting	Description
default_device NAME	The alsa device id to use when none is specified in the URI.
default_format F	The sampling rate, size and channels to use. Wildcards are not allowed. Example - "44100:16:2"
auto_resample yes|no	If set to no, then libasound will not attempt to resample. In this case, the user is responsible for ensuring that the requested sample rate can be produced natively by the device, otherwise an error will occur.
auto_channels yes|no	If set to no, then libasound will not attempt to convert between different channel numbers. The user must ensure that the device supports the requested channels when sampling.
auto_format yes|no	If set to no, then libasound will not attempt to convert between different sample formats (16 bit, 24 bit, floating point, ...). Again the user must ensure that the requested format is available natively from the device.

2.4.2 cdio_paranoia

Plays audio CDs using libcdio. The URI has the form: "cdda://[DEVICE][/TRACK]". The simplest form cdda:// plays the whole disc in the default drive.

Setting	Description
default_byte_order little_endian|big_endian	If the CD drive does not specify a byte order, MPD assumes it is the CPU's native byte order. This setting allows overriding this.
speed N	Request CDParanoia cap the extraction speed to Nx normal CD audio rotation speed, keeping the drive quiet.

2.4.3 curl

Opens remote files or streams over HTTP using libcurl.

Note that unless overridden by the below settings (e.g. by setting them to a blank value), general curl configuration from environment variables such as `http_proxy` or specified in `~/.curlrc` will be in effect.

Setting	Description
proxy	Sets the address of the HTTP proxy server.
proxy_user, proxy_password	Configures proxy authentication.
verify_peer yes|no	Verify the peer's SSL certificate? More information.
verify_host yes|no	Verify the certificate's name against host? More information.

2.4.4 ffmpeg

Access to various network protocols implemented by the FFmpeg library: `gopher://`, `rtp://`, `rtsp://`, `rtmp://`, `rtmpt://`, `rtmps://`

2.4.5 file

Opens local files

2.4.6 mms

Plays streams with the MMS protocol using libmms.

2.4.7 nfs

Allows **MPD** to access files on NFS servers without actually mounting them (i.e. with **libnfs** in userspace, without help from the kernel's VFS layer). All URIs with the `nfs://` scheme are used according to RFC2224. Example:

```
mpc add nfs://servername/path/filename.ogg
```

This plugin uses **libnfs**, which supports only NFS version 3. Since **MPD** is not allowed to bind to so-called "privileged ports", the NFS server needs to enable the `insecure` setting; example `/etc/exports`:

```
/srv/mp3 192.168.1.55(ro,insecure)
```

Don't fear: this will not make your file server insecure; the flag was named a time long ago when privileged ports were thought to be meaningful for security. By today's standards, NFSv3 is not secure at all, and if you believe it is, you're already doomed.

2.4.8 smbclient

Allows **MPD** to access files on SMB/CIFS servers (e.g. Samba or Microsoft Windows). All URIs with the `smb://` scheme are used. Example:

```
mpc add smb://servername/sharename/filename.ogg
mpc add smb://username:password@servername/sharename/filename.ogg
```

2.4.9 qobuz

Play songs from the commercial streaming service Qobuz. It plays URLs in the form `qobuz://track/ID`, e.g.:

```
mpc add qobuz://track/23601296
```

Setting	Description
app_id ID	The Qobuz application id.
app_secret SECRET	The Qobuz application secret.
username USERNAME	The Qobuz user name.
password PASSWORD	The Qobuz password.
format_id N	The Qobuz format identifier, i.e. a number which chooses the format and quality to be requested from Qobuz. The default is "5" (320 kbit/s MP3).

2.5 Decoder plugins

2.5.1 adplug

Decodes AdLib files using libadplug.

Setting	Description
sample_rate	The sample rate that shall be synthesized by the plugin. Defaults to 48000.

2.5.2 audiofile

Decodes WAV and AIFF files using libaudiofile.

2.5.3 faad

Decodes AAC files using libfaad.

2.5.4 ffmpeg

Decodes various codecs using FFmpeg.

Setting	Description
analyzeduration VALUE	Sets the FFmpeg muxer option analyzeduration, which specifies how many microseconds are analyzed to probe the input. The FFmpeg formats documentation has more information.
probesize VALUE	Sets the FFmpeg muxer option probesize, which specifies probing size in bytes, i.e. the size of the data to analyze to get stream information. The FFmpeg formats documentation has more information.

2.5.5 flac

Decodes FLAC files using libFLAC.

2.5.6 dsdiff

Decodes DSDIFF (Direct Stream Digital Interchange File Format) files (`*.dff`). These contain *DSD* instead of PCM.

Setting	Description
lsbitfirst yes|no	Decode the least significant bit first. Default is no.

2.5.7 dsf

Decodes DSF (<https://dsd-guide.com/sites/default/files/white-papers/DSFFileFormatSpec_E.pdf>) files (`*.dsf`). These contain *DSD* instead of PCM.

2.5.8 fluidsynth

MIDI decoder based on FluidSynth.

Setting	Description
sample_rate	The sample rate that shall be synthesized by the plugin. Defaults to 48000.
soundfont	The absolute path of the soundfont file. Defaults to `/usr/share/sounds/sf2/FluidR3_GM.sf2`.

2.5.9 gme

Video game music file emulator based on game-music-emu.

Setting	Description
accuracy yes|no	Enable more accurate sound emulation.
default_fade	The default fade-out time, in seconds. Used by songs that don't specify their own fade-out time.

2.5.10 hybrid_dsd

Hybrid-DSD is an MP4 container file (`*.m4a`) which contains both ALAC and DSD data. It is disabled by default, and works only if you explicitly enable it. Without this plugin, the ALAC parts gets handled by the *FFmpeg decoder plugin*. This plugin should be enabled only if you have a bit-perfect playback path to a DSD-capable DAC; for everybody else, playing back the ALAC copy of the file is better.

2.5.11 mad

Decodes MP3 files using libmad.

2.5.12 mikmod

Module player based on MikMod.

Setting	Description
loop yes|no	Allow backward loops in modules. Default is no.
sample_rate	Sets the sample rate generated by libmikmod. Default is 44100.

2.5.13 modplug

Module player based on MODPlug.

Setting	Description
loop_count	Number of times to loop the module if it uses backward loops. Default is 0 which prevents looping. -1 loops forever.

2.5.14 mpcdec

Decodes Musepack files using libmpcdec.

2.5.15 mpg123

Decodes MP3 files using libmpg123. Currently, this decoder does not support streams (e.g. archived files, remote files over HTTP, ...), only regular local files.

2.5.16 opus

Decodes Opus files using libopus.

2.5.17 pcm

Reads raw PCM samples. It understands the "audio/L16" MIME type with parameters "rate" and "channels" according to RFC 2586. It also understands the MPD-specific MIME type "audio/x-mpd-float".

2.5.18 sidplay

C64 SID decoder based on libsidplayfp or libsidplay2.

Setting	Description
songlength_database PATH	Location of your songlengths file, as distributed with the HVSC. The sidplay plugin checks this for matching MD5 fingerprints. See http://www.hvsc.c64.org/download/C64Music/DOCUMENTS/Songlengths.faq. New songlength format support requires libsidplayfp 2.0 or later.
default_songlength SECONDS	This is the default playing time in seconds for songs not in the songlength database, or in case you're not using a database. A value of 0 means play indefinitely.
default_genre GENRE	Optional default genre for SID songs.
filter yes|no	Turns the SID filter emulation on or off.
kernal	Only libsidplayfp. Roms are not embedded in libsidplayfp - please note https://sourceforge.net/p/sidplay-residfp/news/2013/01/released-libsidplayfp-100beta1/ But some SID tunes require rom images to play. Make C64 rom dumps from your own vintage gear or use rom files from Frodo or VICE emulation software tarballs. Absolute path to kernal rom image file.
basic	Only libsidplayfp. Absolute path to basic rom image file.

2.5.19 sndfile

Decodes WAV and AIFF files using libsndfile.

2.5.20 vorbis

Decodes Ogg-Vorbis files using libvorbis.

2.5.21 wavpack

Decodes WavPack files using libwavpack.

2.5.22 wildmidi

MIDI decoder based on libwildmidi.

Setting	Description
config_file	The absolute path of the timidity config file. Defaults to `/etc/timidity/timidity.cfg`.

2.6 Encoder plugins

2.6.1 flac

Encodes into FLAC (lossless).

Setting	Description
compression	Sets the libFLAC compression level. The levels range from 0 (fastest, least compression) to 8 (slowest, most compression).

2.6.2 lame

Encodes into MP3 using the LAME library.

Setting	Description
quality	Sets the quality for VBR. 0 is the highest quality, 9 is the lowest quality. Cannot be used with bitrate.
bitrate	Sets the bit rate in kilobit per second. Cannot be used with quality.

2.6.3 null

Does not encode anything, passes the input PCM data as-is.

2.6.4 shine

Encodes into MP3 using the Shine library.

Setting	Description
bitrate	Sets the bit rate in kilobit per second.

2.6.5 twolame

Encodes into MP2 using the TwoLAME library.

Setting	Description
quality	Sets the quality for VBR. 0 is the highest quality, 9 is the lowest quality. Cannot be used with bitrate.
bitrate	Sets the bit rate in kilobit per second. Cannot be used with quality.

2.6.6 opus

Encodes into Ogg Opus.

Setting	Description
bitrate	Sets the data rate in bit per second. The special value "auto" lets libopus choose a rate (which is the default), and "max" uses the maximum possible data rate.
complexity	Sets the Opus complexity.
signal	Sets the Opus signal type. Valid values are "auto" (the default), "voice" and "music".
opustags yes|no	Configures how metadata is interleaved into the stream. If set to yes, then metadata is inserted using ogg stream chaining, as specified in **RFC 7845**. If set to no (the default), then ogg stream chaining is avoided and other output-dependent method is used, if available.

2.6.7 vorbis

Encodes into Ogg Vorbis.

Setting	Description
quality	Sets the quality for VBR. -1 is the lowest quality, 10 is the highest quality. Defaults to 3. Cannot be used with bitrate.
bitrate	Sets the bit rate in kilobit per second. Cannot be used with quality.

2.6.8 wave

Encodes into WAV (lossless).

2.7 Resampler plugins

The resampler can be configured in a block named resampler, for example:

```
resampler {
  plugin "soxr"
  quality "very high"
}
```

The following table lists the resampler options valid for all plugins:

Name	Description
plugin	The name of the plugin.

2.7.1 internal

A resampler built into **MPD**. Its quality is very poor, but its CPU usage is low. This is the fallback if **MPD** was compiled without an external resampler.

2.7.2 libsamplerate

A resampler using libsamplerate a.k.a. Secret Rabbit Code (SRC).

Name	Description
type	The interpolator type. Defaults to 2. See below for a list of known types.

The following converter types are provided by libsamplerate:

Type	Description
"Best Sinc Interpolator" or "0"	Band limited sinc interpolation, best quality, 97dB SNR, 96% BW.
"Medium Sinc Interpolator" or "1"	Band limited sinc interpolation, medium quality, 97dB SNR, 90% BW.
"Fastest Sinc Interpolator" or "2"	Band limited sinc interpolation, fastest, 97dB SNR, 80% BW.
"ZOH Sinc Interpolator" or "3"	Zero order hold interpolator, very fast, very poor quality with audible distortions.
"Linear Interpolator" or "4"	Linear interpolator, very fast, poor quality.

2.7.3 soxr

A resampler using libsoxr, the SoX Resampler library

Name	Description
quality	The libsoxr quality setting. Valid values see below.
threads	The number of libsoxr threads. "0" means "automatic". The default is "1" which disables multi-threading.

Valid quality values for libsoxr:

- "very high"
- "high" (the default)
- "medium"
- "low"
- "quick"
- "custom"

If the quality is set to custom also the following settings are available:

Name	Description
precision	The precision in bits. Valid values 16,20,24,28 and 32 bits.
phase_response	Between the 0-100, Where 0=MINIMUM_PHASE and 50=LINEAR_PHASE.
passband_end	The % of source bandwidth where to start filtering. Typical between the 90-99.7.
stopband_begin	The % of the source bandwidth Where the anti aliasing filter start. Value 100+.
attenuation	Reduction in dB's to prevent clipping from the resampling process.
flags	Bitmask with additional option see soxr documentation for specific flags.

2.8 Output plugins

2.8.1 alsa

The Advanced Linux Sound Architecture (ALSA) plugin uses libasound. It is recommended if you are using Linux.

Setting	Description
device NAME	Sets the device which should be used. This can be any valid ALSA device name. The default value is "default", which makes libasound choose a device. It is recommended to use a "hw" or "plughw" device, because otherwise, libasound automatically enables "dmix", which has major disadvantages (fixed sample rate, poor resampler, ...).
buffer_time US	Sets the device's buffer time in microseconds. Don't change unless you know what you're doing.
period_time US	Sets the device's period time in microseconds. Don't change unless you really know what you're doing.
auto_resample yes|no	If set to no, then libasound will not attempt to resample, handing the responsibility over to MPD. It is recommended to let MPD resample (with libsamplerate), because ALSA is quite poor at doing so.
auto_channels yes|no	If set to no, then libasound will not attempt to convert between different channel numbers.
auto_format yes|no	If set to no, then libasound will not attempt to convert between different sample formats (16 bit, 24 bit, floating point, ...).
dop yes|no	If set to yes, then DSD over PCM according to the DoP standard is enabled. This wraps DSD samples in fake 24 bit PCM, and is understood by some DSD capable products, but may be harmful to other hardware. Therefore, the default is no and you can enable the option at your own risk.
allowed_formats F1 F2 ...	Specifies a list of allowed audio formats, separated by a space. All items may contain asterisks as a wild card, and may be followed by "=dop" to enable DoP (DSD over PCM) for this particular format. The first matching format is used, and if none matches, MPD chooses the best fallback of this list. Example: "96000:16:* 192000:24:* dsd64:=*dop *:dsd:*".

The according hardware mixer plugin understands the following settings:

Setting	Description
mixer_device DEVICE	Sets the ALSA mixer device name, defaulting to default which lets ALSA pick a value.
mixer_control NAME	Choose a mixer control, defaulting to PCM. Type amixer scontrols to get a list of available mixer controls.
mixer_index NUMBER	Choose a mixer control index. This is necessary if there is more than one control with the same name. Defaults to 0 (the first one).

The following attributes can be configured at runtime using the outputset command:

Setting	Description
dop 1|0	Allows changing the dop configuration setting at runtime. This takes effect the next time the output is opened.
allowed_formats F1 F2 ...	Allows changing the allowed_formats configuration setting at runtime. This takes effect the next time the output is opened.

2.8.2 ao

The ao plugin uses the portable libao library. Use only if there is no native plugin for your operating system.

Setting	Description
driver D	The libao driver to use for audio output. Possible values depend on what libao drivers are available. See http://www.xiph.org/ao/doc/drivers.html for information on some commonly used drivers. Typical values for Linux include "oss" and "alsa09". The default is "default", which causes libao to select an appropriate plugin.
options O	Options to pass to the selected libao driver.
write_size O	This specifies how many bytes to write to the audio device at once. This parameter is to work around a bug in older versions of libao on sound cards with very small buffers. The default is 1024.

2.8.3 sndio

The sndio plugin uses the sndio library. It should normally be used on OpenBSD.

Setting	Description
device NAME	The audio output device libsndio will attempt to use. The default is "default" which causes libsndio to select the first output device.
buffer_time MS	Set the application buffer time in milliseconds.

2.8.4 fifo

The fifo plugin writes raw PCM data to a FIFO (First In, First Out) file. The data can be read by another program.

Setting	Description
path P	This specifies the path of the FIFO to write to. Must be an absolute path. If the path does not exist, it will be created when MPD is started, and removed when MPD is stopped. The FIFO will be created with the same user and group as MPD is running as. Default permissions can be modified by using the builtin shell command umask. If a FIFO already exists at the specified path it will be reused, and will not be removed when MPD is stopped. You can use the "mkfifo" command to create this, and then you may modify the permissions to your liking.

2.8.5 haiku

Use the SoundPlayer API on the Haiku operating system.

This plugin is unmaintained and contains known bugs. It will be removed soon, unless there is a new maintainer.

2.8.6 jack

The jack plugin connects to a JACK server.

On Windows, this plugin loads `libjack64.dll` at runtime. This means you need to download and install the JACK windows build.

Setting	Description
client_name NAME	The name of the JACK client. Defaults to "Music Player Daemon".
server_name NAME	Optional name of the JACK server.
autostart yes|no	If set to yes, then libjack will automatically launch the JACK daemon. Disabled by default.
source_ports A,B	The names of the JACK source ports to be created. By default, the ports "left" and "right" are created. To use more ports, you have to tweak this option.
destination_ports A,B	The names of the JACK destination ports to connect to.
auto_destination_ports yes|no	If set to *yes*, then MPD will automatically create connections between the send ports of MPD and receive ports of the first sound card; if set to *no*, then MPD will only create connections to the contents of *destination_ports* if it is set. Enabled by default.
ringbuffer_size NBYTES	Sets the size of the ring buffer for each channel. Do not configure this value unless you know what you're doing.

2.8.7 httpd

The httpd plugin creates a HTTP server, similar to ShoutCast / IceCast. HTTP streaming clients like mplayer, VLC, and mpv can connect to it.

It is highly recommended to configure a fixed format, because a stream cannot switch its audio format on-the-fly when the song changes.

Setting	Description
port P	Binds the HTTP server to the specified port.
bind_to_address ADDR	Binds the HTTP server to the specified address (IPv4, IPv6 or local socket). Multiple addresses in parallel are not supported.
encoder NAME	Chooses an encoder plugin. A list of encoder plugins can be found in the encoder plugin reference *Encoder plugins*.
max_clients MC	Sets a limit, number of concurrent clients. When set to 0 no limit will apply.

2.8.8 null

The null plugin does nothing. It discards everything sent to it.

Setting	Description
sync yes|no	If set to no, then the timer is disabled - the device will accept PCM chunks at arbitrary rate (useful for benchmarking). The default behaviour is to play in real time.

2.8.9 oss

The "Open Sound System" plugin is supported on most Unix platforms.

On Linux, OSS has been superseded by ALSA. Use the ALSA output plugin *alsa* instead of this one on Linux.

Setting	Description
device PATH	Sets the path of the PCM device. If not specified, then MPD will attempt to open /dev/sound/dsp and /dev/dsp.

The according hardware mixer plugin understands the following settings:

Setting	Description
mixer_device DEVICE	Sets the OSS mixer device path, defaulting to /dev/mixer.
mixer_control NAME	Choose a mixer control, defaulting to PCM.

2.8.10 openal

The "OpenAL" plugin uses libopenal. It is supported on many platforms. Use only if there is no native plugin for your operating system.

Setting	Description
device NAME	Sets the device which should be used. This can be any valid OpenAL device name. If not specified, then libopenal will choose a default device.

2.8.11 osx

The "Mac OS X" plugin uses Apple's CoreAudio API.

Setting	Description
device NAME	Sets the device which should be used. Uses device names as listed in the "Audio Devices" window of "Audio MIDI Setup".
hog_device yes|no	Hog the device. This means that it takes exclusive control of the audio output device it is playing through, and no other program can access it.
dop yes|no	If set to yes, then DSD over PCM according to the DoP standard is enabled. This wraps DSD samples in fake 24 bit PCM, and is understood by some DSD capable products, but may be harmful to other hardware. Therefore, the default is no and you can enable the option at your own risk. Under macOS you must make sure to select a physical mode on the output device which supports at least 24 bits per channel as the Mac OS X plugin only changes the sample rate.
channel_map SOURCE,SOURCE,...	Specifies a channel map. If your audio device has more than two outputs this allows you to route audio to auxillary outputs. For predictable results you should also specify a "format" with a fixed number of channels, e.g. "*::*2". The number of items in the channel map must match the number of output channels of your output device. Each list entry specifies the source for that output channel; use "-1" to silence an output. For example, if you have a four-channel output device and you wish to send stereo sound (format "*::*2") to outputs 3 and 4 while leaving outputs 1 and 2 silent then set the channel map to "-1,-1,0,1". In this example '0' and '1' denote the left and right channel respectively. The channel map may not refer to outputs that do not exist according to the format. If the format is "::1" (mono) and you have a four-channel sound card then "-1,-1,0,0" (dual mono output on the second pair of sound card outputs) is a valid channel map but "-1,-1,0,1" is not because the second channel ('1') does not exist when the output is mono.

2.8.12 pipe

The pipe plugin starts a program and writes raw PCM data into its standard input.

Setting	Description
command CMD	This command is invoked with the shell.

2.8.13 pulse

The pulse plugin connects to a PulseAudio server. Requires libpulse.

Setting	Description
server HOSTNAME	Sets the host name of the PulseAudio server. By default, **MPD** connects to the local PulseAudio server.
sink NAME	Specifies the name of the PulseAudio sink **MPD** should play on.
media_role ROLE	Specifies a custom media role that **MPD** reports to PulseAudio. Default is "music". (optional).
scale_volume FACTOR	Specifies a linear scaling coefficient (ranging from 0.5 to 5.0) to apply when adjusting volume through **MPD**. For example, chosing a factor equal to `"0.7"` means that setting the volume to 100 in **MPD** will set the PulseAudio volume to 70%, and a factor equal to `"3.5"` means that volume 100 in **MPD** corresponds to a 350% PulseAudio volume.

2.8.14 recorder

The recorder plugin writes the audio played by **MPD** to a file. This may be useful for recording radio streams.

Setting	Description
path P	Write to this file.
format_path P	An alternative to path which provides a format string referring to tag values. The special tag iso8601 emits the current date and time in ISO8601 format (UTC). Every time a new song starts or a new tag gets received from a radio station, a new file is opened. If the format does not render a file name, nothing is recorded. A tag name enclosed in percent signs ('%') is replaced with the tag value. Example: `-/.mpd/recorder/%artist% - %title%.ogg`. Square brackets can be used to group a substring. If none of the tags referred in the group can be found, the whole group is omitted. Example: [-/.mpd/recorder/[%artist% -]%title%.ogg] (this omits the dash when no artist tag exists; if title also doesn't exist, no file is written). The operators "|" (logical "or") and "&" (logical "and") can be used to select portions of the format string depending on the existing tag values. Example: -/.mpd/recorder/[%title%|%name%].ogg (use the "name" tag if no title exists)
encoder NAME	Chooses an encoder plugin. A list of encoder plugins can be found in the encoder plugin reference *Encoder plugins*.

2.8.15 shout

The shout plugin connects to a ShoutCast or IceCast server using libshout. It forwards tags to this server.

You must set a format.

Setting	Description
host HOSTNAME	Sets the host name of the ShoutCast / IceCast server.
port PORTNUMBER	Connect to this port number on the specified host.
timeout SECONDS	Set the timeout for the shout connection in seconds. Defaults to 2 seconds.
protocol icecast2|icecast1|shoutcast	Specifies the protocol that wil be used to connect to the server. The default is "icecast2".
tls disabled|auto|auto_no_plain|rfc2818|rfc2817	Specifies what kind of TLS to use. The default is "disabled" (no TLS).
mount URI	Mounts the **MPD** stream in the specified URI.
user USERNAME	Sets the user name for submitting the stream to the server. Default is "source".
password PWD	Sets the password for submitting the stream to the server.
name NAME	Sets the name of the stream.
genre GENRE	Sets the genre of the stream (optional).
description DESCRIPTION	Sets a short description of the stream (optional).
url URL	Sets a URL associated with the stream (optional).
public yes|no	Specifies whether the stream should be "public". Default is no.
encoder PLUGIN	Chooses an encoder plugin. Default is vorbis *vorbis*. A list of encoder plugins can be found in the encoder plugin reference *Encoder plugins*.

2.8.16 sles

Plugin using the OpenSL ES audio API. Its primary use is local playback on Android, where *ALSA* is not available. It supports 16 bit and floating point samples.

2.8.17 solaris

The "Solaris" plugin runs only on SUN Solaris, and plays via /dev/audio.

Setting	Description
device PATH	Sets the path of the audio device, defaults to /dev/audio.

2.8.18 wasapi

The Windows Audio Session API plugin uses WASAPI, which is supported started from Windows Vista. It is recommended if you are using Windows.

Setting	Description
device NAME	Sets the device which should be used. This can be any valid audio device name, or index number. The default value is "", which makes WASAPI choose the default output device.
enumerate yes|no	Enumerate all devices in log while playing started. Useful for device configuration. The default value is "no".
exclusive yes|no	Exclusive mode blocks all other audio source, and get best audio quality without resampling. Stopping playing release the exclusive control of the output device. The default value is "no".
dop yes|no	Enable DSD over PCM. Require exclusive mode. The default value is "no".

2.9 Filter plugins

2.9.1 ffmpeg

Configures a FFmpeg filter graph.

This plugin requires building with `libavfilter` (FFmpeg).

Setting	Description
graph "…"	Specifies the `libavfilter` graph; read the FFmpeg documentation for details

2.9.2 hdcd

Decode HDCD.

This plugin requires building with `libavfilter` (FFmpeg).

2.9.3 normalize

Normalize the volume during playback (at the expense of quality).

2.9.4 null

A no-op filter. Audio data is returned as-is.

2.9.5 route

Reroute channels.

Setting	Description
routes "0>0, 1>1, …"	Specifies the channel mapping.

2.10 Playlist plugins

2.10.1 asx

Reads `.asx` playlist files.

2.10.2 cue

Reads `.cue` files.

2.10.3 embcue

Reads CUE sheets from the `CUESHEET` tag of song files.

2.10.4 m3u

Reads `.m3u` playlist files.

2.10.5 extm3u

Reads extended `.m3u` playlist files.

2.10.6 flac

Reads the cuesheet metablock from a FLAC file.

2.10.7 pls

Reads `.pls` playlist files.

2.10.8 rss

Reads music links from `.rss` files.

2.10.9 soundcloud

Download playlist from SoundCloud. It accepts URIs starting with soundcloud://.

Setting	Description
apikey KEY	An API key to access the SoundCloud servers.

2.10.10 xspf

Reads XSPF playlist files.

2.11 Archive plugins

2.11.1 bz2

Allows to load single bzip2 compressed files using libbz2. Does not support seeking.

2.11.2 zzip

Allows to load music files from ZIP archives using zziplib.

2.11.3 iso

Allows to load music files from ISO 9660 images using libcdio.

CHAPTER

THREE

DEVELOPER'S MANUAL

3.1 Introduction

This is a guide for those who wish to hack on the MPD source code. MPD is an open project, and we are always happy about contributions. So far, more than 150 people have contributed patches. This document is work in progress. Most of it may be incomplete yet. Please help!

3.2 Code Style

- indent with tabs (width 8)
- don't write CPP when you can write C++: use inline functions and constexpr instead of macros
- comment your code, document your APIs
- the code should be C++17 compliant, and must compile with **GCC** 8 and **clang** 5
- all code must be exception-safe
- classes and functions names use CamelCase; variables are lower-case with words separated by underscore

Some example code:

```
Foo(const char *abc, int xyz)
{
    if (abc == nullptr) {
            LogWarning("Foo happened!");
            return -1;
    }

    return xyz;
}
```

3.2.1 Error handling

If an error occurs, throw a C++ exception, preferably derived from `std::runtime_error`. The function's API documentation should mention that. If a function cannot throw exceptions, add `noexcept` to its prototype.

Some parts of MPD use callbacks to report completion; the handler classes usually have an "error" callback which receives a `std::exception_ptr` (e.g. `BufferedSocket::OnSocketError()`). Wrapping errors in `std::exception_ptr` allows propagating details about the error across thread boundaries to the entity which is interested in handling it (e.g. giving the MPD client details about an I/O error caught by the decoder thread).

Out-of-memory errors (i.e. `std::bad_alloc`) do not need to be handled. Some operating systems such as Linux do not report out-of-memory to userspace, and instead kill a process to recover. Even if we know we are out of memory, there is little we can do except for aborting the process quickly. Any other attempts to give back memory may cause page faults on the way which make the situation worse.

Error conditions which are caused by a bug do not need to be handled at runtime; instead, use `assert()` to detect them in debug builds.

3.3 git Branches

There are two active branches in the git repository:

- the "unstable" branch called `master` where new features are merged. This will become the next major release eventually.
- the "stable" branch (currently called `v0.22.x`) where only bug fixes are merged.

Once **MPD** 0.23 is released, a new branch called `v0.23.x` will be created for 0.23 bug-fix releases; after that, `v0.22.x` will eventually cease to be maintained.

After bug fixes have been added to the "stable" branch, it will be merged into `master`. This ensures that all known bugs are fixed in all active branches.

3.4 Hacking The Source

MPD sources are managed in a git repository on Github.

Always write your code against the latest git:

```
git clone git://github.com/MusicPlayerDaemon/MPD
```

If you already have a clone, update it:

```
git pull --rebase git://github.com/MusicPlayerDaemon/MPD master
```

You can do without `--rebase`, but we recommend that you rebase your repository on the "master" repository all the time.

Configure with the option `--werror`. Enable as many plugins as possible, to be sure that you don't break any disabled code.

Don't mix several changes in one single patch. Create a separate patch for every change. Tools like **stgit** help you with that. This way, we can review your patches more easily, and we can pick the patches we like most first.

3.4.1 Basic stgit usage

stgit allows you to create a set of patches and refine all of them: you can go back to any patch at any time, and re-edit it (both the code and the commit message). You can reorder patches and insert new patches at any position. It encourages creating separate patches for tiny changes.

stgit needs to be initialized on a git repository:

```
stg init
```

Before you edit the code, create a patch:

```
stg new my-patch-name
```

stgit now asks you for the commit message.

Now edit the code. Once you're finished, you have to "refresh" the patch, i.e. your edits are incorporated into the patch you have created:

```
stg refresh
```

You may now continue editing the same patch, and refresh it as often as you like. Create more patches, edit and refresh them.

To view the list of patches, type stg series. To go back to a specific patch, type stg goto my-patch-name; now you can re-edit it (don't forget stg refresh when you're finished with that patch).

When the whole patch series is finished, convert stgit patches to git commits:

```
stg commit
```

3.5 Submitting Patches

Submit pull requests on GitHub: https://github.com/MusicPlayerDaemon/MPD/pulls

CHAPTER

FOUR

PROTOCOL

4.1 General protocol syntax

4.1.1 Protocol overview

The **MPD** command protocol exchanges line-based text records between client and server over TCP. Once the client is connected to the server, they conduct a conversation until the client closes the connection. The conversation flow is always initiated by the client.

All data between the client and the server is encoded in UTF-8.

The client transmits a command sequence, terminated by the newline character `\n`. The server will respond with one or more lines, the last of which will be a completion code.

When the client connects to the server, the server will answer with the following line:

```
OK MPD version
```

where `version` is a version identifier such as 0.12.2. This version identifier is the version of the protocol spoken, not the real version of the daemon. (There is no way to retrieve this real version identifier from the connection.)

4.1.2 Requests

```
COMMAND [ARG...]
```

If arguments contain spaces, they should be surrounded by double quotation marks.

Argument strings are separated from the command and any other arguments by linear white-space (' ' or '\t').

4.1.3 Responses

A command returns `OK` on completion or `ACK some error` on failure. These denote the end of command execution.

Some commands return more data before the response ends with `OK`. Each line is usually in the form `NAME: VALUE`. Example:

```
foo: bar
OK
```

Binary Responses

Some commands can return binary data. This is initiated by a line containing `binary: 1234` (followed as usual by a newline). After that, the specified number of bytes of binary data follows, then a newline, and finally the `OK` line.

If the object to be transmitted is large, the server may choose a reasonable chunk size and transmit only a portion. The maximum chunk size can be changed by clients with the *binarylimit* command.

Usually, the response also contains a `size` line which specifies the total (uncropped) size, and the command usually has a way to specify an offset into the object; this way, the client can copy the whole file without blocking the connection for too long.

Example:

```
foo: bar
binary: 42
<42 bytes>
OK
```

Failure responses

The nature of the error can be gleaned from the information that follows the `ACK`. `ACK` lines are of the form:

```
ACK [error@command_listNum] {current_command} message_text
```

These responses are generated by a call to `commandError`. They contain four separate terms. Let's look at each of them:

- `error`: numeric value of one of the `ACK_ERROR` constants defined in *src/protocol/Ack.hxx*.
- `command_listNum`: offset of the command that caused the error in a *Command List*. An error will always cause a command list to terminate at the command that causes the error.
- `current_command`: name of the command, in a *Command List*, that was executing when the error occurred.
- `message_text`: some (hopefully) informative text that describes the nature of the error.

An example might help. Consider the following sequence sent from the client to the server:

```
command_list_begin
volume 86
play 10240
status
command_list_end
```

The server responds with:

```
ACK [50@1] {play} song doesn't exist: "10240"
```

This tells us that the play command, which was the second in the list (the first or only command is numbered 0), failed with error 50. The number 50 translates to `ACK_ERROR_NO_EXIST` – the song doesn't exist. This is reiterated by the message text which also tells us which song doesn't exist.

4.1.4 Command lists

To facilitate faster adding of files etc. you can pass a list of commands all at once using a command list. The command list begins with *command_list_begin* or *command_list_ok_begin* and ends with *command_list_end*.

It does not execute any commands until the list has ended. The response is a concatentation of all individual responses. On success for all commands, `OK` is returned. If a command fails, no more commands are executed and the appropriate `ACK` error is returned. If *command_list_ok_begin* is used, `list_OK` is returned for each successful command executed in the command list.

4.1.5 Ranges

Some commands (e.g. *delete*) allow specifying a range in the form `START:END` (the `END` item is not included in the range, similar to ranges in the Python programming language). If `END` is omitted, then the maximum possible value is assumed.

4.1.6 Filters

All commands which search for songs (e.g. *find* and *searchadd*) share a common filter syntax:

```
find EXPRESSION
```

`EXPRESSION` is a string enclosed in parentheses which can be one of:

- `(TAG == 'VALUE')`: match a tag value; if there are multiple values of the given type, at least one must match. `(TAG != 'VALUE')`: mismatch a tag value; if there are multiple values of the given type, none of them must match. The special tag `any` checks all tag types. `AlbumArtist` looks for `VALUE` in `AlbumArtist` and falls back to `Artist` tags if `AlbumArtist` does not exist. `VALUE` is what to find. An empty value string means: match only if the given tag type does not exist at all; this implies that negation with an empty value checks for the existence of the given tag type.
- `(TAG contains 'VALUE')` checks if the given value is a substring of the tag value.
- `(TAG =~ 'VALUE')` and `(TAG !~ 'VALUE')` use a Perl-compatible regular expression instead of doing a simple string comparison. (This feature is only available if **MPD** was compiled with `libpcre`)
- `(file == 'VALUE')`: match the full song URI (relative to the music directory).
- `(base 'VALUE')`: restrict the search to songs in the given directory (relative to the music directory).
- `(modified-since 'VALUE')`: compares the file's time stamp with the given value (ISO 8601 or UNIX time stamp).
- `(AudioFormat == 'SAMPLERATE:BITS:CHANNELS')`: compares the audio format with the given value. See *Global Audio Format* for a detailed explanation.
- `(AudioFormat =~ 'SAMPLERATE:BITS:CHANNELS')`: matches the audio format with the given mask (i.e. one or more attributes may be *).
- `(!EXPRESSION)`: negate an expression. Note that each expression must be enclosed in parentheses, e.g. `(!(artist == 'VALUE'))` (which is equivalent to `(artist != 'VALUE')`)
- `(EXPRESSION1 AND EXPRESSION2 ...)`: combine two or more expressions with logical "and". Note that each expression must be enclosed in parentheses, e.g. `((artist == 'FOO') AND (album == 'BAR'))`

The **find** commands are case sensitive, while **search** and related commands ignore case.

Prior to MPD 0.21, the syntax looked like this:

```
find TYPE VALUE
```

Escaping String Values

String values are quoted with single or double quotes, and special characters within those values must be escaped with the backslash (\). Keep in mind that the backslash is also the escape character on the protocol level, which means you may need to use double backslash.

Example expression which matches an artist named `foo'bar"`:

```
(Artist == "foo\'bar\"")
```

At the protocol level, the command must look like this:

```
find "(Artist == \"foo\\'bar\\\"\")"
```

The double quotes enclosing the artist name must be escaped because they are inside a double-quoted `find` parameter. The single quote inside that artist name must be escaped with two backslashes; one to escape the single quote, and another one because the backslash inside the string inside the parameter needs to be escaped as well. The double quote has three confusing backslashes: two to build one backslash, and another one to escape the double quote on the protocol level. Phew!

To reduce confusion, you should use a library such as libmpdclient which escapes command arguments for you.

4.1.7 Tags

The following tags are supported by **MPD**:

- **artist**: the artist name. Its meaning is not well-defined; see "*composer*" and "*performer*" for more specific tags.
- **artistsort**: same as artist, but for sorting. This usually omits prefixes such as "The".
- **album**: the album name.
- **albumsort**: same as album, but for sorting.
- **albumartist**: on multi-artist albums, this is the artist name which shall be used for the whole album. The exact meaning of this tag is not well-defined.
- **albumartistsort**: same as albumartist, but for sorting.
- **title**: the song title.
- **track**: the decimal track number within the album.
- **name**: a name for this song. This is not the song title. The exact meaning of this tag is not well-defined. It is often used by badly configured internet radio stations with broken tags to squeeze both the artist name and the song title in one tag.
- **genre**: the music genre.
- **date**: the song's release date. This is usually a 4-digit year.
- **originaldate**: the song's original release date.
- **composer**: the artist who composed the song.
- **performer**: the artist who performed the song.
- **conductor**: the conductor who conducted the song.

- **work**: "a work is a distinct intellectual or artistic creation, which can be expressed in the form of one or more audio recordings"
- **grouping**: "used if the sound belongs to a larger category of sounds/music" (from the IDv2.4.0 TIT1 description).
- **comment**: a human-readable comment about this song. The exact meaning of this tag is not well-defined.
- **disc**: the decimal disc number in a multi-disc album.
- **label**: the name of the label or publisher.
- **musicbrainz_artistid**: the artist id in the MusicBrainz database.
- **musicbrainz_albumid**: the album id in the MusicBrainz database.
- **musicbrainz_albumartistid**: the album artist id in the MusicBrainz database.
- **musicbrainz_trackid**: the track id in the MusicBrainz database.
- **musicbrainz_releasetrackid**: the release track id in the MusicBrainz database.
- **musicbrainz_workid**: the work id in the MusicBrainz database.

There can be multiple values for some of these tags. For example, **MPD** may return multiple lines with a `performer` tag. A tag value is a UTF-8 string.

4.1.8 Other Metadata

The response to *lsinfo* and similar commands may contain *song tags* and other metadata, specifically:

- `duration`: the duration of the song in seconds; may contain a fractional part.
- `time`: like `duration`, but as integer value. This is deprecated and is only here for compatibility with older clients. Do not use.
- `Range`: if this is a queue item referring only to a portion of the song file, then this attribute contains the time range in the form `START-END` or `START-` (open ended); both `START` and `END` are time stamps within the song in seconds (may contain a fractional part). Example: `60-120` plays only the second minute; "`180` skips the first three minutes.
- `Format`: the audio format of the song (or an approximation to a format supported by MPD and the decoder plugin being used). When playing this file, the `audio` value in the *status* response should be the same.
- `Last-Modified`: the time stamp of the last modification of the underlying file in ISO 8601 format. Example: "*2008-09-28T20:04:57Z*"

4.2 Recipes

4.2.1 Queuing

Often, users run **MPD** with *random* enabled, but want to be able to insert songs "before" the rest of the playlist. That is commonly called "queuing".

MPD implements this by allowing the client to specify a "priority" for each song in the playlist (commands *priod* and *priodid*). A higher priority means that the song is going to be played before the other songs.

In "random" mode, **MPD** maintains an internal randomized sequence of songs. In this sequence, songs with a higher priority come first, and all songs with the same priority are shuffled (by default, all songs are shuffled, because all have the same priority "0"). When you increase the priority of a song, it is moved to the front of the sequence according to its new priority, but always after the current one. A song that has been played already (it's "before" the current song

in that sequence) will only be scheduled for repeated playback if its priority has become bigger than the priority of the current song. Decreasing the priority of a song will move it farther to the end of the sequence. Changing the priority of the current song has no effect on the sequence. During playback, a song's priority is reset to zero.

4.3 Command reference

Note: For manipulating playlists and playing, there are two sets of commands. One set uses the song id of a song in the playlist, while another set uses the playlist position of the song. The commands using song ids should be used instead of the commands that manipulate and control playback based on playlist position. Using song ids is a safer method when multiple clients are interacting with **MPD**.

4.3.1 Querying MPD's status

clearerror Clears the current error message in status (this is also accomplished by any command that starts playback).

currentsong Displays the song info of the current song (same song that is identified in status). Information about the current song is represented by key-value pairs, one on each line. The first pair must be the *file* key-value pair.

idle [SUBSYSTEMS...][1] Waits until there is a noteworthy change in one or more of **MPD**'s subsystems. As soon as there is one, it lists all changed systems in a line in the format `changed: SUBSYSTEM`, where SUBSYSTEM is one of the following:

- `database`: the song database has been modified after *update*.
- `update`: a database update has started or finished. If the database was modified during the update, the `database` event is also emitted.
- `stored_playlist`: a stored playlist has been modified, renamed, created or deleted
- `playlist`: the queue (i.e. the current playlist) has been modified
- `player`: the player has been started, stopped or seeked or tags of the currently playing song have changed (e.g. received from stream)
- `mixer`: the volume has been changed
- `output`: an audio output has been added, removed or modified (e.g. renamed, enabled or disabled)
- `options`: options like repeat, random, crossfade, replay gain
- `partition`: a partition was added, removed or changed
- `sticker`: the sticker database has been modified.
- `subscription`: a client has subscribed or unsubscribed to a channel
- `message`: a message was received on a channel this client is subscribed to; this event is only emitted when the queue is empty
- `neighbor`: a neighbor was found or lost
- `mount`: the mount list has changed

[1] Since **MPD** 0.14

Change events accumulate, even while the connection is not in "idle" mode; no events get lost while the client is doing something else with the connection. If an event had already occurred since the last call, the new *idle* command will return immediately.

While a client is waiting for *idle* results, the server disables timeouts, allowing a client to wait for events as long as mpd runs. The *idle* command can be canceled by sending the command *noidle* (no other commands are allowed). **MPD** will then leave *idle* mode and print results immediately; might be empty at this time. If the optional `SUBSYSTEMS` argument is used, **MPD** will only send notifications when something changed in one of the specified subsytems.

status Reports the current status of the player and the volume level.

- `partition`: the name of the current partition (see *Partition commands*)
- `volume`: `0-100` (deprecated: `-1` if the volume cannot be determined)
- `repeat`: `0` or `1`
- `random`: `0` or `1`
- `single`[2]: `0`, `1`, or `oneshot`[6]
- `consume`[?]: `0` or `1`
- `playlist`: 31-bit unsigned integer, the playlist version number
- `playlistlength`: integer, the length of the playlist
- `state`: `play`, `stop`, or `pause`
- `song`: playlist song number of the current song stopped on or playing
- `songid`: playlist songid of the current song stopped on or playing
- `nextsong`[?]: playlist song number of the next song to be played
- `nextsongid`[?]: playlist songid of the next song to be played
- `time`: total time elapsed (of current playing/paused song) in seconds (deprecated, use `elapsed` instead)
- `elapsed`[3]: Total time elapsed within the current song in seconds, but with higher resolution.
- `duration`[5]: Duration of the current song in seconds.
- `bitrate`: instantaneous bitrate in kbps
- `xfade`: `crossfade` in seconds
- `mixrampdb`: `mixramp` threshold in dB
- `mixrampdelay`: `mixrampdelay` in seconds
- `audio`: The format emitted by the decoder plugin during playback, format: `samplerate:bits:channels`. See *Global Audio Format* for a detailed explanation.
- `updating_db`: `job id`
- `error`: if there is an error, returns message here

MPD may omit lines which have no (known) value. Older **MPD** versions used to have a "magic" value for "unknown", e.g. "`volume: -1`".

stats Displays statistics.

[2] Since **MPD** 0.15
[6] Since **MPD** 0.21
[3] Since **MPD** 0.16
[5] Since **MPD** 0.20

- `artists`: number of artists
- `albums`: number of albums
- `songs`: number of songs
- `uptime`: daemon uptime in seconds
- `db_playtime`: sum of all song times in the database in seconds
- `db_update`: last db update in UNIX time (seconds since 1970-01-01 UTC)
- `playtime`: time length of music played

4.3.2 Playback options

consume {STATE}? Sets consume state to `STATE`, `STATE` should be 0 or 1. When consume is activated, each song played is removed from playlist.

crossfade {SECONDS} Sets crossfading between songs.

mixrampdb {deciBels} Sets the threshold at which songs will be overlapped. Like crossfading but doesn't fade the track volume, just overlaps. The songs need to have MixRamp tags added by an external tool. 0dB is the normalized maximum volume so use negative values, I prefer -17dB. In the absence of mixramp tags crossfading will be used. See http://sourceforge.net/projects/mixramp

mixrampdelay {SECONDS} Additional time subtracted from the overlap calculated by mixrampdb. A value of "nan" disables MixRamp overlapping and falls back to crossfading.

random {STATE} Sets random state to `STATE`, `STATE` should be 0 or 1.

repeat {STATE} Sets repeat state to `STATE`, `STATE` should be 0 or 1.

setvol {VOL} Sets volume to `VOL`, the range of volume is 0-100.

single {STATE}? Sets single state to `STATE`, `STATE` should be `0`, `1` or `oneshot`?. When single is activated, playback is stopped after current song, or song is repeated if the 'repeat' mode is enabled.

replay_gain_mode {MODE}? Sets the replay gain mode. One of `off`, `track`, `album`, `auto` . Changing the mode during playback may take several seconds, because the new settings do not affect the buffered data. This command triggers the `options` idle event.

replay_gain_status Prints replay gain options. Currently, only the variable `replay_gain_mode` is returned.

volume {CHANGE} Changes volume by amount `CHANGE`. Deprecated, use *setvol* instead.

4.3.3 Controlling playback

next Plays next song in the playlist.

pause {STATE} Pause or resume playback. Pass 1 to pause playback or `0` to resume playback. Without the parameter, the pause state is toggled.

play [SONGPOS] Begins playing the playlist at song number `SONGPOS`.

playid [SONGID] Begins playing the playlist at song `SONGID`.

previous Plays previous song in the playlist.

seek {SONGPOS} {TIME} Seeks to the position `TIME` (in seconds; fractions allowed) of entry `SONGPOS` in the playlist.

seekid {SONGID} {TIME} Seeks to the position `TIME` (in seconds; fractions allowed) of song `SONGID`.

seekcur {TIME} Seeks to the position `TIME` (in seconds; fractions allowed) within the current song. If prefixed by + or -, then the time is relative to the current playing position.

stop Stops playing.

4.3.4 The Queue

Note: The "queue" used to be called "current playlist" or just "playlist", but that was deemed confusing, because "playlists" are also files containing a sequence of songs. Those "playlist files" or "stored playlists" can be *loaded into the queue* and the queue can be *saved into a playlist file*, but they are not to be confused with the queue.

Many of the command names in this section reflect the old naming convention, but for the sake of compatibility, we cannot rename commands.

There are two ways to address songs within the queue: by their position and by their id.

The position is a 0-based index. It is unstable by design: if you move, delete or insert songs, all following indices will change, and a client can never be sure what song is behind a given index/position.

Song ids on the other hand are stable: an id is assigned to a song when it is added, and will stay the same, no matter how much it is moved around. Adding the same song twice will assign different ids to them, and a deleted-and-readded song will have a new id. This way, a client can always be sure the correct song is being used.

Many commands come in two flavors, one for each address type. Whenever possible, ids should be used.

add {URI} Adds the file `URI` to the playlist (directories add recursively). `URI` can also be a single file.

Clients that are connected via local socket may add arbitrary local files (URI is an absolute path). Example:

```
add "/home/foo/Music/bar.ogg"
```

addid {URI} [POSITION] Adds a song to the playlist (non-recursive) and returns the song id. `URI` is always a single file or URL. For example:

```
addid "foo.mp3"
Id: 999
OK
```

clear Clears the queue.

delete [{POS} | {START:END}] Deletes a song from the playlist.

deleteid {SONGID} Deletes the song `SONGID` from the playlist

move [{FROM} | {START:END}] {TO} Moves the song at `FROM` or range of songs at `START:END`[?] to `TO` in the playlist.

moveid {FROM} {TO} Moves the song with `FROM` (songid) to `TO` (playlist index) in the playlist. If `TO` is negative, it is relative to the current song in the playlist (if there is one).

playlist

Displays the queue.

Do not use this, instead use *playlistinfo*.

playlistfind {FILTER} Finds songs in the queue with strict matching.

playlistid {SONGID} Displays a list of songs in the playlist. `SONGID` is optional and specifies a single song to display info for.

playlistinfo [[SONGPOS] | [START:END]] Displays a list of all songs in the playlist, or if the optional argument is given, displays information only for the song SONGPOS or the range of songs START:END[7]

playlistsearch {FILTER} Searches case-insensitively for partial matches in the queue.

plchanges {VERSION} [START:END] Displays changed songs currently in the playlist since VERSION. Start and end positions may be given to limit the output to changes in the given range.

To detect songs that were deleted at the end of the playlist, use playlistlength returned by status command.

plchangesposid {VERSION} [START:END] Displays changed songs currently in the playlist since VERSION. This function only returns the position and the id of the changed song, not the complete metadata. This is more bandwidth efficient.

To detect songs that were deleted at the end of the playlist, use playlistlength returned by status command.

prio {PRIORITY} {START:END...} Set the priority of the specified songs. A higher priority means that it will be played first when "random" mode is enabled.

A priority is an integer between 0 and 255. The default priority of new songs is 0.

prioid {PRIORITY} {ID...} Same as *priod*, but address the songs with their id.

rangeid {ID} {START:END}[4] Since **MPD** 0.19 Specifies the portion of the song that shall be played. START and END are offsets in seconds (fractional seconds allowed); both are optional. Omitting both (i.e. sending just ":") means "remove the range, play everything". A song that is currently playing cannot be manipulated this way.

shuffle [START:END] Shuffles the queue. START:END is optional and specifies a range of songs.

swap {SONG1} {SONG2} Swaps the positions of SONG1 and SONG2.

swapid {SONG1} {SONG2} Swaps the positions of SONG1 and SONG2 (both song ids).

addtagid {SONGID} {TAG} {VALUE} Adds a tag to the specified song. Editing song tags is only possible for remote songs. This change is volatile: it may be overwritten by tags received from the server, and the data is gone when the song gets removed from the queue.

cleartagid {SONGID} [TAG] Removes tags from the specified song. If TAG is not specified, then all tag values will be removed. Editing song tags is only possible for remote songs.

4.3.5 Stored playlists

Playlists are stored inside the configured playlist directory. They are addressed with their file name (without the directory and without the *.m3u* suffix).

Some of the commands described in this section can be used to run playlist plugins instead of the hard-coded simple *m3u* parser. They can access playlists in the music directory (relative path including the suffix), playlists in arbitrary location (absolute path including the suffix; allowed only for clients that are connected via local socket), or remote playlists (absolute URI with a supported scheme).

listplaylist {NAME} Lists the songs in the playlist. Playlist plugins are supported.

listplaylistinfo {NAME} Lists the songs with metadata in the playlist. Playlist plugins are supported.

listplaylists Prints a list of the playlist directory. After each playlist name the server sends its last modification time as attribute "Last-Modified" in ISO 8601 format. To avoid problems due to clock differences between clients and the server, clients should not compare this value with their local clock.

load {NAME} [START:END] Loads the playlist into the current queue. Playlist plugins are supported. A range may be specified to load only a part of the playlist.

playlistadd {NAME} {URI} Adds URI to the playlist *NAME.m3u*. *NAME.m3u* will be created if it does not exist.

[4] Since **MPD** 0.19

playlistclear {NAME} Clears the playlist *NAME.m3u*.

playlistdelete {NAME} {SONGPOS} Deletes `SONGPOS` from the playlist *NAME.m3u*.

playlistmove {NAME} {FROM} {TO} Moves the song at position `FROM` in the playlist *NAME.m3u* to the position `TO`.

rename {NAME} {NEW_NAME} Renames the playlist *NAME.m3u* to *NEW_NAME.m3u*.

rm {NAME} Removes the playlist *NAME.m3u* from the playlist directory.

save {NAME} Saves the queue to *NAME.m3u* in the playlist directory.

4.3.6 The music database

albumart {URI} {OFFSET} Locate album art for the given song and return a chunk of an album art image file at offset `OFFSET`.

This is currently implemented by searching the directory the file resides in for a file called `cover.png`, `cover.jpg`, `cover.tiff` or `cover.bmp`.

Returns the file size and actual number of bytes read at the requested offset, followed by the chunk requested as raw bytes (see *Binary Responses*), then a newline and the completion code.

Example:

```
albumart foo/bar.ogg 0
size: 1024768
binary: 8192
<8192 bytes>
OK
```

count {FILTER} [group {GROUPTYPE}] Count the number of songs and their total playtime in the database matching `FILTER` (see *Filters*). The following prints the number of songs whose title matches "Echoes":

```
count title Echoes
```

The *group* keyword may be used to group the results by a tag. The first following example prints per-artist counts while the next prints the number of songs whose title matches "Echoes" grouped by artist:

```
count group artist
count title Echoes group artist
```

A group with an empty value contains counts of matching songs which don't have this group tag. It exists only if at least one such song is found.

getfingerprint {URI}

Calculate the song's audio fingerprint. Example (abbreviated fingerprint):

```
getfingerprint "foo/bar.ogg"
chromaprint: AQACcEmSREmWJJmkIT_6CCf64...
OK
```

This command is only available if MPD was built with `libchromaprint` (`-Dchromaprint=enabled`).

find {FILTER} [sort {TYPE}] [window {START:END}] Search the database for songs matching `FILTER` (see *Filters*).

`sort` sorts the result by the specified tag. The sort is descending if the tag is prefixed with a minus ('-'). Without `sort`, the order is undefined. Only the first tag value will be used, if multiple of the same type exist. To sort by "Artist", "Album" or "AlbumArtist", you should specify "ArtistSort", "AlbumSort" or "AlbumArtistSort" instead. These will automatically fall back to the former if "*Sort" doesn't exist. "AlbumArtist" falls back to just "Artist". The type "Last-Modified" can sort by file modification time.

`window` can be used to query only a portion of the real response. The parameter is two zero-based record numbers; a start number and an end number.

findadd {FILTER} [sort {TYPE}] [window {START:END}] Search the database for songs matching `FILTER` (see *Filters*) and add them to the queue. Parameters have the same meaning as for *find*.

list {TYPE} {FILTER} [group {GROUPTYPE}] Lists unique tags values of the specified type. `TYPE` can be any tag supported by **MPD**.

Additional arguments may specify a *filter*. The *group* keyword may be used (repeatedly) to group the results by one or more tags.

The following example lists all album names, grouped by their respective (album) artist:

```
list album group albumartist
```

`list file` was implemented in an early **MPD** version, but does not appear to make a lot of sense. It still works (to avoid breaking compatibility), but is deprecated.

listall [URI] Lists all songs and directories in `URI`.

Do not use this command. Do not manage a client-side copy of **MPD**'s database. That is fragile and adds huge overhead. It will break with large databases. Instead, query **MPD** whenever you need something.

listallinfo [URI] Same as *listall*, except it also returns metadata info in the same format as *lsinfo*

Do not use this command. Do not manage a client-side copy of **MPD**'s database. That is fragile and adds huge overhead. It will break with large databases. Instead, query **MPD** whenever you need something.

listfiles {URI} Lists the contents of the directory `URI`, including files are not recognized by **MPD**. `URI` can be a path relative to the music directory or an URI understood by one of the storage plugins. The response contains at least one line for each directory entry with the prefix "file: " or "directory: ", and may be followed by file attributes such as "Last-Modified" and "size".

For example, "smb://SERVER" returns a list of all shares on the given SMB/CIFS server; "nfs://servername/path" obtains a directory listing from the NFS server.

lsinfo [URI] Lists the contents of the directory `URI`. The response contains records starting with `file`, `directory` or `playlist`, each followed by metadata (*tags* or *other metadata*).

When listing the root directory, this currently returns the list of stored playlists. This behavior is deprecated; use "listplaylists" instead.

This command may be used to list metadata of remote files (e.g. URI beginning with "http://" or "smb://").

Clients that are connected via local socket may use this command to read the tags of an arbitrary local file (URI is an absolute path).

readcomments {URI} Read "comments" (i.e. key-value pairs) from the file specified by "URI". This "URI" can be a path relative to the music directory or an absolute path.

This command may be used to list metadata of remote files (e.g. URI beginning with "http://" or "smb://").

The response consists of lines in the form "KEY: VALUE". Comments with suspicious characters (e.g. newlines) are ignored silently.

The meaning of these depends on the codec, and not all decoder plugins support it. For example, on Ogg files, this lists the Vorbis comments.

readpicture {URI} {OFFSET} Locate a picture for the given song and return a chunk of the image file at offset OFFSET. This is usually implemented by reading embedded pictures from binary tags (e.g. ID3v2's APIC tag).

Returns the following values:

- size: the total file size
- type: the file's MIME type (optional)
- binary: see *Binary Responses*

If the song file was recognized, but there is no picture, the response is successful, but is otherwise empty.

Example:

```
readpicture foo/bar.ogg 0
size: 1024768
type: image/jpeg
binary: 8192
<8192 bytes>
OK
```

search {FILTER} [sort {TYPE}] [window {START:END}] Search the database for songs matching FILTER (see *Filters*). Parameters have the same meaning as for *find*, except that search is not case sensitive.

searchadd {FILTER} [sort {TYPE}] [window {START:END}] Search the database for songs matching FILTER (see *Filters*) and add them to the queue.

Parameters have the same meaning as for *search*.

searchaddpl {NAME} {FILTER} [sort {TYPE}] [window {START:END}] Search the database for songs matching FILTER (see *Filters*) and add them to the playlist named NAME.

If a playlist by that name doesn't exist it is created.

Parameters have the same meaning as for *search*.

update [URI] Updates the music database: find new files, remove deleted files, update modified files.

URI is a particular directory or song/file to update. If you do not specify it, everything is updated.

Prints updating_db: JOBID where JOBID is a positive number identifying the update job. You can read the current job id in the *status* response.

rescan [URI] Same as *update*, but also rescans unmodified files.

4.3.7 Mounts and neighbors

A "storage" provides access to files in a directory tree. The most basic storage plugin is the "local" storage plugin which accesses the local file system, and there are plugins to access NFS and SMB servers.

Multiple storages can be "mounted" together, similar to the *mount* command on many operating systems, but without cooperation from the kernel. No superuser privileges are necessary, because this mapping exists only inside the **MPD** process.

mount {PATH} {URI} Mount the specified remote storage URI at the given path. Example:

```
mount foo nfs://192.168.1.4/export/mp3
```

unmount {PATH} Unmounts the specified path. Example:

```
unmount foo
```

listmounts Queries a list of all mounts. By default, this contains just the configured `music_directory`. Example:

```
listmounts
mount:
storage: /home/foo/music
mount: foo
storage: nfs://192.168.1.4/export/mp3
OK
```

listneighbors Queries a list of "neighbors" (e.g. accessible file servers on the local net). Items on that list may be used with the *mount* command. Example:

```
listneighbors
neighbor: smb://FOO
name: FOO (Samba 4.1.11-Debian)
OK
```

4.3.8 Stickers

"Stickers"[7] are pieces of information attached to existing **MPD** objects (e.g. song files, directories, albums; but currently, they are only implemented for song). Clients can create arbitrary name/value pairs. **MPD** itself does not assume any special meaning in them.

The goal is to allow clients to share additional (possibly dynamic) information about songs, which is neither stored on the client (not available to other clients), nor stored in the song files (**MPD** has no write access).

Client developers should create a standard for common sticker names, to ensure interoperability.

Objects which may have stickers are addressed by their object type ("song" for song objects) and their URI (the path within the database for songs).

sticker get {TYPE} {URI} {NAME} Reads a sticker value for the specified object.

sticker set {TYPE} {URI} {NAME} {VALUE} Adds a sticker value to the specified object. If a sticker item with that name already exists, it is replaced.

sticker delete {TYPE} {URI} [NAME] Deletes a sticker value from the specified object. If you do not specify a sticker name, all sticker values are deleted.

sticker list {TYPE} {URI} Lists the stickers for the specified object.

sticker find {TYPE} {URI} {NAME} Searches the sticker database for stickers with the specified name, below the specified directory (URI). For each matching song, it prints the URI and that one sticker's value.

sticker find {TYPE} {URI} {NAME} = {VALUE} Searches for stickers with the given value.

Other supported operators are: "<", ">"

4.3.9 Connection settings

close Closes the connection to **MPD**. **MPD** will try to send the remaining output buffer before it actually closes the connection, but that cannot be guaranteed. This command will not generate a response.

Clients should not use this command; instead, they should just close the socket.

kill Kills **MPD**.

Do not use this command. Send SIGTERM to **MPD** instead, or better: let your service manager handle **MPD** shutdown (e.g. **systemctl stop mpd**).

password {PASSWORD} This is used for authentication with the server. PASSWORD is simply the plaintext password.

ping Does nothing but return "OK".

binarylimit SIZE[7]

Set the maximum *binary response* size for the current connection to the specified number of bytes.

A bigger value means less overhead for transmitting large entities, but it also means that the connection is blocked for a longer time.

tagtypes Shows a list of available tag types. It is an intersection of the metadata_to_use setting and this client's tag mask.

About the tag mask: each client can decide to disable any number of tag types, which will be omitted from responses to this client. That is a good idea, because it makes responses smaller. The following tagtypes sub commands configure this list.

tagtypes disable {NAME...} Remove one or more tags from the list of tag types the client is interested in. These will be omitted from responses to this client.

tagtypes enable {NAME...} Re-enable one or more tags from the list of tag types for this client. These will no longer be hidden from responses to this client.

tagtypes clear Clear the list of tag types this client is interested in. This means that **MPD** will not send any tags to this client.

tagtypes all Announce that this client is interested in all tag types. This is the default setting for new clients.

4.3.10 Partition commands

These commands allow a client to inspect and manage "partitions". A partition is one frontend of a multi-player MPD process: it has separate queue, player and outputs. A client is assigned to one partition at a time.

partition {NAME} Switch the client to a different partition.

listpartitions Print a list of partitions. Each partition starts with a partition keyword and the partition's name, followed by information about the partition.

newpartition {NAME} Create a new partition.

delpartition {NAME} Delete a partition. The partition must be empty (no connected clients and no outputs).

moveoutput {OUTPUTNAME} Move an output to the current partition.

[7] Since **MPD** 0.22.4

4.3.11 Audio output devices

disableoutput {ID} Turns an output off.

enableoutput {ID} Turns an output on.

toggleoutput {ID} Turns an output on or off, depending on the current state.

outputs Shows information about all outputs.

```
outputid: 0
outputname: My ALSA Device
plugin: alsa
outputenabled: 0
attribute: dop=0
OK
```

Return information:

- `outputid`: ID of the output. May change between executions
- `outputname`: Name of the output. It can be any.
- `outputenabled`: Status of the output. 0 if disabled, 1 if enabled.

outputset {ID} {NAME} {VALUE} Set a runtime attribute. These are specific to the output plugin, and supported values are usually printed in the *outputs* response.

4.3.12 Reflection

config Dumps configuration values that may be interesting for the client. This command is only permitted to "local" clients (connected via local socket).

The following response attributes are available:

- `music_directory`: The absolute path of the music directory.

commands Shows which commands the current user has access to.

notcommands Shows which commands the current user does not have access to.

urlhandlers Gets a list of available URL handlers.

decoders Print a list of decoder plugins, followed by their supported suffixes and MIME types. Example response:

```
plugin: mad
suffix: mp3
suffix: mp2
mime_type: audio/mpeg
plugin: mpcdec
suffix: mpc
```

4.3.13 Client to client

Clients can communicate with each others over "channels". A channel is created by a client subscribing to it. More than one client can be subscribed to a channel at a time; all of them will receive the messages which get sent to it.

Each time a client subscribes or unsubscribes, the global idle event `subscription` is generated. In conjunction with the *channels* command, this may be used to auto-detect clients providing additional services.

New messages are indicated by the `message` idle event.

If your MPD instance has multiple partitions, note that client-to-client messages are local to the current partition.

subscribe {NAME} Subscribe to a channel. The channel is created if it does not exist already. The name may consist of alphanumeric ASCII characters plus underscore, dash, dot and colon.

unsubscribe {NAME} Unsubscribe from a channel.

channels Obtain a list of all channels. The response is a list of "channel:" lines.

readmessages Reads messages for this client. The response is a list of "channel:" and "message:" lines.

sendmessage {CHANNEL} {TEXT} Send a message to the specified channel.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- search

INDEX